Contrôler la visibilité des aspects avec Aspectboxes

Alexandre Bergel

RMoD Team, INRIA - Lille Nord Europe - USTL - CNRS UMR 8022, Lille, France
http://www.bergel.eu

Résumé. La composition et l'intéraction des aspects est un domaine de recherche très actif. Bien que plusieurs solutions existent, telles que l'agencement des aspects et des advices, les approches proposées par des langages à aspects supposent qu'une connaissance générale des aspects soit nécessaire pour pouvoir les composer, et même ceci ne permet pas d'éviter les interactions implicites résultant d'une composition.
Cet article présente les aspectboxes, un mécanisme de visibilité pour aspects. L'unité élémentaire de visibilité est un aspectbox. Un aspectbox encapsule des définitions d'aspects. Un aspectbox peut être utilisé par d'autres aspectboxes pour aider la construction incrémentale de logiciel à base d'aspects. Une classe peut utiliser un aspectbox dans le but de bénéficier des aspects définis.

1 Introduction

La programmation par aspects définit des constructions linguistiques permettant une meilleure modularité complétant les mécanismes traditionnels d'héritage dans les langages de programmation à objets tels que Java et C++. Un aspect définit un comportement transversal à des hiérarchies de classes (Kiczales et al., 2001) ou de composition de fonctions (Dutchyn et al., 2006; Gal et al., 2001).

Un des challenges de la programmation orientée par aspects est la composition d'aspects (Douence et al., 2004; Klaeren et al., 2000; Nagy et al., 2005; Brichau et al., 2002; Tanter, 2008; Havinga et al., 2006; Marot et Wuyts, 2008). L'ordonnancement d'aspects, de points de jointure et d'advices, est couramment utilisé (Kiczales et al., 2001; Tanter, 2006). Bien que la plupart des langages à aspects offrent de tels mécanismes de composition, une mise en pratique d'un grand nombre d'aspects s'avère être une tâche complexe (Lopez-Herrejon et al., 2006). Une des principales difficultés à composer des aspects réside dans l'identification et le contrôle des interactions implicites résultant de cette composition.

En considérant un aspect comme une extension d'un système de base, apporter de multiples extensions à un même système peut engendrer des interactions involontaires, même si ces extensions ont été développées de façon indépendante. Nous pensons que la cause de cette situation est le manque d'un mécanisme de visibilité pour la programmation par aspects.

Associer une visibilité aux aspects a essentiellement deux caractéristiques :

– Lorsqu'ils ont des visibilités différentes, les aspects n'ont pas à être composés puisqu'ils ne sont pas visibles les uns des autres ;

– Plusieurs aspects dans une même visibilité peuvent potentiellement avoir des advices en conflits. Ces aspects doivent donc être composés de façon traditionnelle en les ordonnant.

Plusieurs mécanismes de visibilité et de portée pour aspects ont été proposés (Rajan et Sullivan, 2003; Tanter, 2008; Hirschfeld et al., 2008; Dutchyn et al., 2006; Aracic et al., 2006). Cependant, ces visibilités sont intrinsèquement liées au flot de contrôle et à la pile d'exécution. Le but de cet article est de poser les fondations pour un mécanisme simple et intuitif de visibilité pour aspects, indépendant de tout paramètre statique.

Le mécanisme des *aspectboxes* est un système de visibilité pour aspects. Chaque aspect a une visibilité définie par un aspectbox. Un aspectbox peut être utilisé par un autre aspectbox. Dans ce cas, la visibilité définie par ce second aspectbox est étendue par ce premier. L'"effet" d'un aspect est limité par l'aspectbox qui le définit et les aspectboxes qui l'utilisent. En dehors de cet aspectbox ou des aspectboxes utilisateurs, c'est comme si aucun aspect n'était défini. De par l'utilisation des aspectboxes, des aspects apportés par des aspectboxes différents ne peuvent être en conflit.

Le principal attrait des aspectboxes est de remettre en question la nécessité de composition si celle-ci n'est pas réalisable ou nécessaire. L'idée est ainsi d'éviter une composition si un conflit n'a pas lieu d'être.

Dans cet article, un scénario illustrant l'un des problèmes liés à la composition d'aspects sera présenté (section 2). Le mécanisme des aspectboxes sera ensuite décrite et ses propriétés énumérées (section 3). Enfin, préalablement à la conclusion (section 5) les travaux connexes liés aux aspectboxes et à la composition d'aspects seront évoqués (section 4).

2 Composition d'aspects et dépendances implicites

La composition d'aspects est une question cruciale dont la communauté s'efforce de définir les implications (Klaeren et al., 2000; Nagy et al., 2005; Brichau et al., 2002), notamment les interactions inter-aspects (Douence et al., 2004) et la sémantique de composition (Tanter, 2006).

La composition d'aspects peut générer des dépendances cachées (Douence et al., 2004). Appliquer un aspect à un système peut perturber le fonctionnement d'autres aspects et d'une façon générale, l'application dans son ensemble. Cette section décrit et analyse un tel scénario en utilisant un exemple inspiré par Charfi *et al.* (Charfi et al., 2006).

2.1 Description d'un agenda

Application de base. La classe Agenda contient la liste de rendez-vous d'un utilisateur et offre des méthodes d'ajout et de suppression. La classe Agenda et l'aspect Notification sont illustrés par la figure 1. En utilisant la syntaxe Java, la définition d'Agenda est :

```
public class Agenda {
    LinkedList<String> meetings = new LinkedList<String>() ;
    public void addMeeting (String meeting) { this.meetings.add(meeting) ; }
    public void removeMeeting (String meeting) { this.meetings.remove(meeting) ; }
    public void clear() {
        for(String meeting : new LinkedList<String>(meetings))
            meetings.remove(meeting) ;
}}
```

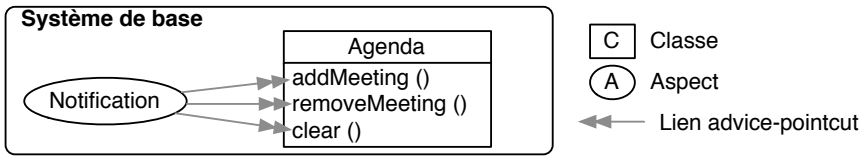

FIG. 1 – *Application Agenda composée d'une classe Agenda est d'un aspect Notification.*

Une représentation visuelle d'un agenda doit être mise à jour quand l'agenda est modifié. Une façon naturelle de définir une notification est d'utiliser un aspect. Un aspect Notification définit une mise à jour quand un rendez-vous est ajouté ou supprimé. En utilisant une syntaxe AspectJ[1], le code source de cet aspect est :

```
aspect Notification {
    // Intercepte les appels de addMeeting() et removeMeeting()
    // sans etre appeles par clear ()
    after () :                                          // Premier advice
        !cflow (call (* Agenda.clear())) &&
        (call (* Agenda.removeMeeting(..)) || call (* Agenda.addMeeting(..))) {
        Display.update () ;
    }

    // Intercepte les appels a clear()
    after() : call (* Agenda.clear()) {                 // Deuxieme advice
        Display.update() ;
}}
```

Les appels à addMeeting(String) et removeMeeting(String) provoquent une mise à jour seulement s'ils ne sont pas émis par la méthode clear() (premier advice). Une fois que clear () a enlevé tous les éléments de la collection meetings, une mise à jour est effectuée (deuxième advice).

Une utilisation de la classe Agenda est illustrée par la classe Application :

```
public class Application {
    public static void main(String[] args) {
        Agenda agenda = new Agenda() ;
        agenda.addMeeting("02/06/2009 - Meeting at INRIA") ;
        agenda.addMeeting("05/06/2009 - Dinner party") ;
        agenda.addMeeting("23/03/2009 - My birthday") ;
        agenda.clear() ;
}}
```

L'exécution de la méthode main invoque 4 fois la méthode update() sur la classe Display représentant 3 ajouts de rendez-vous, suivi d'un appel à la méthode clear(), cette dernière appelant 3 fois la méthode removeMeeting(String).

[1] http://www.eclipse.org/aspectj

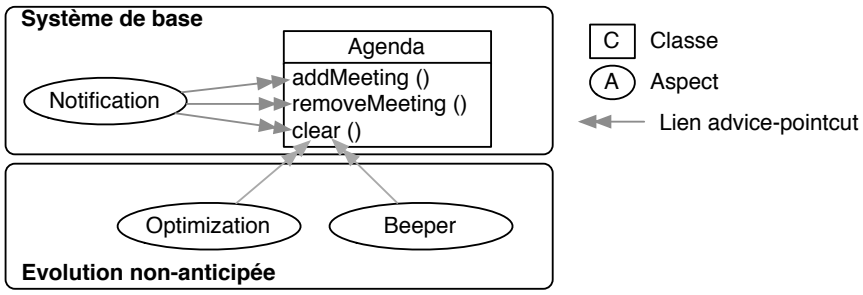

FIG. 2 – *Application Agenda raffinée avec deux nouveaux aspects, Optimization et Beeper.*

Évolution non anticipée. L'application Agenda est fonctionnelle ; cependant, dû à la volonté de certains utilisateurs, l'effacement de la liste des rendez-vous doit être optimisé et un effet sonore doit être ajouté lorsque cet effacement se produit. La méthode clear() a un coût élevé : une copie de la liste des rendez-vous est effectuée et les rendez-vous sont enlevés un à un.

Ces deux évolutions sont définies par des aspects. Deux nouveaux aspects sont ajoutés à notre application. La figure 2 schématise cette nouvelle version. Une optimisation est décrite par l'aspect Optimization :

```
aspect Optimization {
    void around (Agenda agenda) : target (agenda) && call (* Agenda.clear()) {
        agenda.meetings = new LinkedList() ;
}}
```

Ensuite, un effet sonore est émis par un aspect Beeper :

```
aspect Beeper {
    void around () : call (* Agenda.clear ()) {
        proceed () ;
        Beeper.emitBeep () ;
}}
```

L'application *Agenda* est désormais composée de trois classes (Agenda, Display et Application) et de trois aspects (Notification, Optimization et Beeper).

2.2 Analyse

Plusieurs advices pour un même point de jointure. Les aspects Optimization et Beeper définissent chacun un advice de type *around* sur la méthode clear () de la classe Agenda. Un advice de type *around* est exécuté à la place du point de jointure (la méthode clear () dans ce cas précis). Le point de jointure d'origine peut être invoqué par l'utilisation de l'élément syntaxique proceed (...). Les aspects Optimization et Beeper ont pour effet de remplacer la méthode clear (). De par l'utilisation de proceed (...), l'advice apporté par Beeper appelle la version d'origine de clear (). De plus, l'aspect Notification définit un advice pour la méthode clear ().

La méthode clear () est instrumentée avec trois advices apportés par Notification, Optimization et Beeper. Aucune règle de composition n'est définie, ces trois aspects ne sont pas ordonnés. En conséquence, le résultat de la composition finale n'est pas explicite et est ainsi dépendante du *weaver*. Si l'aspect Optimization est installé *avant* Beeper, alors l'invocation de clear () émettra un son et utilisera l'optimisation. Cependant si Beeper est installé avant Optimization, l'expression Beeper.emitBeep () ne sera pas executée car l'advice apporté par Optimization remplacera celui de Beeper.

Bien que AspectJ propose un mécanisme pour ordonner les aspects (declare precedence : ...), son utilisation n'est pas obligatoire et nécessite une connaissance avancée des aspects mis en jeu pour pouvoir être utilisés.

3 Donner une visibilité aux aspects avec les aspectboxes

3.1 Aspectboxes en bref

Aspectboxes est un mécanisme de visibilité pour aspects. Un aspect est défini dans un aspectbox et le comportement apporté par cet aspect est limité à (i) cet aspectbox et (ii) aux aspectboxes qui utilisent cet aspectbox. Une classe peut utiliser un ou plusieurs aspectboxes dans le but de bénéficier du comportement des aspects. Les advices apportés par les aspects contenus dans les aspectboxes utilisés par une classe ont une portée limitée à la classe les utilisant. Les advices de ces aspects ne seront donc exécutés que pour les appels de méthode contenus dans cette classe.

La visibilité d'un aspect est *statique*. L'utilisation ou non d'un aspectbox par une classe régit l'activation d'un aspect pour tous les appels de méthode effectués dans cette classe (*i.e.*, syntaxiquement situés dans la définition de cette classe).

On définit la notation suivante : un aspectbox A_2 utilisant un autre aspectbox A_1 s'écrit $A_2 \rhd A_1$. La relation $\rhd$ est transitive et asymétrique. Si $A_2 \rhd A_1$ et $A_3 \rhd A_2$, alors on a $A_3 \rhd A_1$. L'asymétrie de $\rhd$ permet de ne pas avoir de cycle dans les graphes d'utilisation d'aspectboxes.

Un aspectbox A_2 qui utilise un autre aspectbox A_1 donne "priorité" aux aspects définis dans A_2 sur ceux de A_1. Les aspects apportés par A_2 s'appliquent *après* ceux de A_1. Cette relation de priorité est donc définie au niveau des aspectboxes, et pas au niveau des aspects.

Il est important de préciser qu'un point de jointure contenu dans un aspect peut référencer n'importe quelle classe, indépendamment du fait que celles-ci utilisent ou non les aspectboxes définissant ces aspects.

L'application Agenda utilisant les aspectboxes est décrite par la figure 3. Agenda est instrumentée par 3 aspects, Notification, Optimization, et Beeper. Chacun de ces aspects est dans un aspectbox distinct. Les différentes variations de la classe Agenda sont simultanément accessibles depuis les aspectboxes.

Les sections suivantes décrivent les propriétés et les bénéfices d'un tel mécanisme. On utilise une syntaxe Smalltalk en raison de l'utilisation par notre prototype d'AspectS (Hirschfeld, 2003).

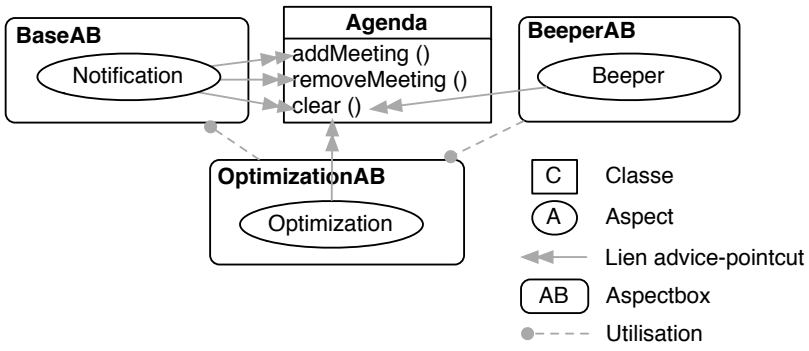

FIG. 3 – *Différentes variantes de Agenda.*

3.2 Espace de noms pour aspects

L'aspectbox OptimizationAB définit l'aspect Optimization :

```
(Aspectbox named : #OptimizationAB)
    createAspectNamed : #Optimization
```

L'aspect Optimization instrumente la classe Agenda. L'advice qui optimise la méthode clear est défini de la façon suivante :

```
OptimizationAB. Optimization>>adviceOptimitizationOfClear
    ↑ AroundAdvice
        pointcut : (JoinPointDescriptor
                                targetClass : Agenda targetSelector : #clear)
        aroundBlock : [ :receiver :arguments :aspect |
                                "receiver est donc une instance de la classe Agenda"
                                receiver setMeetings : OrderedCollection new]
```

La définition de l'aspect Optimization est similaire au code AspectJ (cf. section 2.1). La méthode clear est le point de coupure. L'expression receiver setMeetings : OrderedCollection new affecte une collection vide à la variable meetings.

Plusieurs aspects appelés Optimization peuvent coexister s'ils appartiennent à des aspectboxes différents. On peut noter que la classe Agenda, utilisée dans Optimization, n'a pas besoin d'utiliser l'aspectbox OptimizationAB. Un aspect peut référencer n'importe quelle classe du système.

3.3 Raffinement incrémental

L'aspectbox BeeperAB utilise OptimizationAB. L'optimisation sur clear est donc utilisée depuis BeeperAB. Du point de vue de BeeperAB, cela s'apparente à la situation selon laquelle aucune aspect n'était défini sur Agenda :

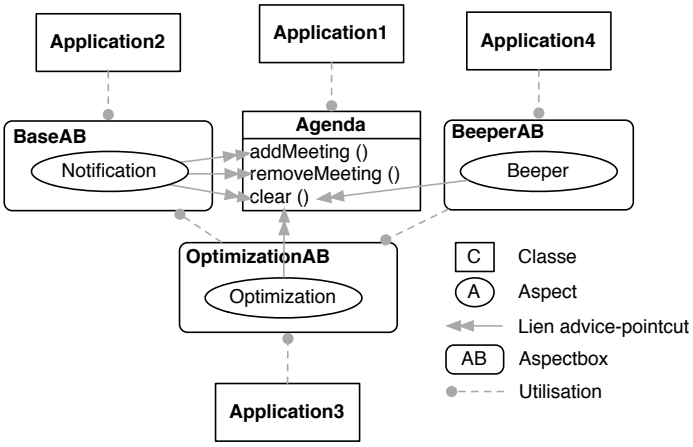

FIG. 4 – *Déclinaison de la classe Agenda en 4 versions.*

```
(Aspectbox named : #BeeperAB)
    createAspectNamed : #Beeper ;
    use : #OptimizationAB

BeeperAB. Beeper>>adviceBeeper
    ↑ AroundAdvice
        pointcut : (JoinPointDescriptor
                                targetClass : Agenda targetSelector : #clear)
        aroundBlock : [ :receiver :arguments :aspect |
                                aspect proceed.
                                Beeper emitBeep]
```

L'aspect Beeper a donc priorité sur Optimization puisque BeeperAB utilise OptimizationAB. Il instrumente donc la version optimisée de la classe Agenda. Invoquer la méthode clear depuis l'aspectbox BeeperAB utilise donc les advices définis dans ces deux aspects.

Chaque aspectbox peut potentiellement définir une nouvelle version des classes instrumentées. Ces versions sont accessibles *simultanément* par différents clients. La figure 4 montre 4 classes Application utilisant une version différente de Agenda : Application1 utilise la classe Agenda originelle, sans aucun aspect ; Application2 utilise Agenda avec la notification ; Application3 bénéficie de l'optimisation et de la notification ; Application4 utilise les trois aspects, le beeper, l'optimisation et la notification.

3.4 Composition des aspectboxes

La figure 5 décrit une composition des aspectboxes plus complexe que celles déjà présentées. Un nouveau aspectbox TracerAB a été ajouté. Il instrumente les classes Agenda et AgendaManager. AgendaFactory référence Agenda et utilise deux aspectboxes, OptimizationAB et TracerAB. La classe Application reference AgendaFactory, et appartient à l'aspectbox BeeperAB.

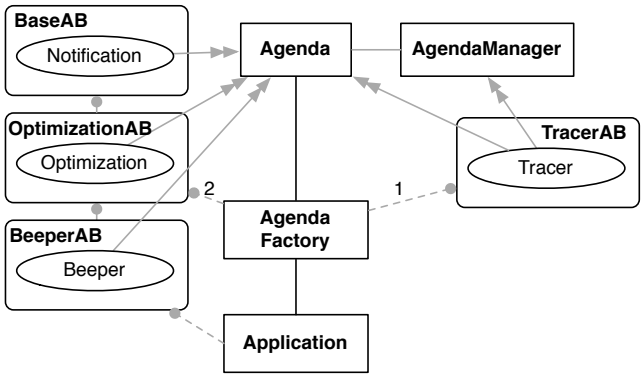

FIG. 5 – *La classe Agenda est déclinée en 4 versions.*

La classe AgendaFactory utilise deux aspectboxes qui instrumentent une même classe, Agenda. Il est donc nécessaire pour AgendaFactory d'ordonner Optimization et Tracer. Ceci est représenté sur la figure 5 par les valeurs numériques sur les relations d'utilisation : l'accroissement numérique reflète l'ordonnancement. L'aspect Tracer est ainsi prioritaire sur Optimization.

Il est important de préciser que les appels de méthodes effectués sur les instances de Agenda par la classe AgendaFactory utilisent les aspects Tracer, Opmitization, et Notification. Ceci est la conséquence de l'utilisation de OptimizationAB et TracerAB par AgendaFactory.

Cependant, les appels de méthodes contenus dans la classe Application sur les instances de AgendaFactory utilisent Beeper, Optimization, et Notification. Ceci est illustré par les exemples suivants :

```
AgendaFactory»getExampleAgenda
    | agenda |
    agenda := Agenda new.
    agenda addMeeting : ('02/06/2009 - Meeting at INRIA').
    ↑ agenda

AgendaFactory»emptyAllAgenda
    self agendas do : [ :agenda |agenda clear]

Application»main
    | factory agenda |
    factory := AgendaFactory new.
    agenda := factory getExampleAgenda.
    agenda clear.
    factory emptyAllAgenda.
```

Cet exemple montre les différentes visibilités mises en jeu. La méthode getExampleAgenda utilise la notification et la trace lors de son appel à addMeeting. La méthode AgendaFactory»emptyAllAgenda bénificie de l'optimisation. Cependant, la génération de trace n'est pas utilisée lors des appels sur Agenda par Application»main puisque Application n'utilise pas TracerAB.

3.5 Résumé des propriétés des aspectboxes

La version de l'application Agenda utilisant les aspectbox (figure 2) a les propriétés suivantes :

- *Evitement des conflits entre aspects.* L'utilisation d'un mécanisme de visibilité permet de limiter le champ d'application d'un aspect. Les conflits de composition pouvant apparaître entre différents aspects sont évités à partir du moment où ces aspects sont définis dans des aspectboxes différents.
- *Multiples versions d'un système.* Par l'utilisation d'un mécanisme de visibilité, plusieurs versions d'un système sont accessibles de façon concurrente.
- *Extension minimale du langage d'aspects.* Le mécanisme des aspectboxes définit un espace de noms pour les aspects, les classes et l'application d'aspects.
- *Indépendance de la pile d'appels de méthode.* L'activation d'aspects avec les aspectboxes est indépendante de la pile d'appels de méthode. D'après notre expérience, avoir un comportement dépendant de la pile d'exécution est difficile à comprendre et s'adapte mal aux applications multi-thread, en combinaison avec l'utilisation d'une interface graphique et des serveurs réseaux.

4 Travaux connexes

AspectJ. Le langage pour décrire les points de coupure avec AspectJ[2] offre un mécanisme pour restreindre la déclaration d'un point de coupure à un package Java ou à une classe (*i.e.*, les primitives within et withincode). L'intérêt de ces constructions consiste à restreindre le domaine d'application des points de jointure.

Cependant, les advices définissant les instrumentations sont visibles globalement. Les primitives de restriction pour la déclaration de points de coupure ne peuvent être utilisées à limiter la visibilité d'un aspect, puisque celle-ci sera toujours globale.

CaesarJ. Une unification des aspects, des packages et des classes est proposée par CaesarJ (Aracic et al., 2006). Un aspect peut être déployé globalement ou localement à un *thread*.

Les aspectboxes offrent un mécanisme syntaxique de visibilité : un aspect est visible dans l'aspectbox qui le définit. Avec CaesarJ, un aspect est visible au *thread* qui ordonne son déploiement.

Stratégie de déploiement. Tanter propose un nouveau mécanisme appelé stratégie de déploiement (Tanter, 2008) qui donne un contrôle sur (i) la propagation d'un aspect en fonction de la pile d'exécution, (ii) la propagation parmi les objets, et (iii) une particularisation des pointcuts au moment du déploiement.

Un aspectbox n'a pas vocation à être installé dynamiquement. Seule la dimension (iii) identifiée par Tanter exprime donc la portée des aspectboxes.

Classboxes. Le système de modules des classboxes permet à une classe d'être étendue par ajout ou redéfinition de méthode. Ces extensions sont visibles dans une portée locale bien

[2] http://eclipse.org/aspectj

définie. Plusieurs extensions d'un même ensemble de classes peuvent coexister dans un même programme (Bergel et al., 2005).

Les classboxes et aspectboxes ont une racine commune qui est l'utilisation d'un mécanisme de visibilité pour limiter l'impact d'une extension. Alors que les classboxes permettent un raffinement structurel (*i.e.*, addition et redéfinition de membres de classe), les aspectboxes permettent un raffinement comportemental (*i.e.*, utilisation d'aspects).

Raffinement de classes. Privat et Ducournau (2005) propose un système de modules avec raffinement de classes dans le langage PRM[3]. Des similarités existent entre PRM et aspectboxes. Par exemple, ces deux méchanismes permettent la définition d'une hierarchie de modules dans laquelle les classes peuvent être raffinées. PRM offre un méta-modèle très générique constitué d'entités globales et locale. Ce méta-modèle est ensuite décliné en deux systèmes, un à classe et l'autre à modules.

Il semblerait que les aspectboxes puissent être exprimées dans PRM. Cependant, Une analyse destiné à certifier ceci est nécessaire.

5 Conclusion

La composition d'aspect s'avère être un problème difficile rempli d'embûches : un ordonnancement des aspects ou des advices permet d'avoir un contrôle affiné de leur composition ; mais une connaissance générale des aspects est nécessaire. De plus, cette composition peut créer des interactions implicites.

La composition d'aspects proposée par AspectJ suppose que *tous les aspects formant une application doivent être accessibles à tous les clients de cette application*. L'approche décrite dans cet article se base sur une hypothèse différente : *l'utilisation d'un ou plusieurs aspects peut être limitée à un ensemble bien défini d'utilisateurs*.

Un des grands bénéfices des aspectboxes est d'éviter les conflits entre aspects si ces aspects ne sont pas accessibles par les mêmes clients.

Une large adoption de la programmation orientée par aspects repose sur la facilité d'utilisation et de la mise en place de ce paradigme. Alors que la composition d'aspects peut s'avérer particulièrement subtile, faciliter cette composition peut simplifier considérablement l'utilisation d'un grand nombre d'aspects.

Remerciements. Nous remercions Marie-Hélène Bergel-Peyre, Marie Boudiguet, Stéphane Ducasse, Bachir Ghoubali, Jannik Laval et Aline Senart pour leurs précieuses relectures.

Références

Aracic, I., V. Gasiunas, M. Mezini, et K. Ostermann (2006). An overview of CaesarJ. *Transactions on Aspect-Oriented Software Development 3880*, 135 – 173.

Bergel, A., S. Ducasse, O. Nierstrasz, et R. Wuyts (2005). Classboxes : Controlling visibility of class extensions. *Journal of Computer Languages, Systems and Structures 31*(3-4), 107–126.

[3] http://www.lirmm.fr/prm

Brichau, J., K. Mens, et K. D. Volder (2002). Building composable aspect-specific languages with logic metaprogramming. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2002)*, Volume 2487 of *LNCS*. Springer-Verlag.

Charfi, A., M. Riveill, M. Blay-Fornarino, et A.-M. Pinna-Dery (2006). Transparent and dynamic aspect composition. In *In Proceedings of the 4th Software Engineering Properties of Languages and Aspect Technologies (SPLAT) Workshop*.

Douence, R., P. Fradet, et M. Südholt (2004). Composition, reuse and interaction analysis of stateful aspects. In *AOSD '04 : Proceedings of the 3rd international conference on Aspect-oriented software development*, New York, NY, USA, pp. 141–150. ACM Press.

Dutchyn, C., D. B. Tucker, et S. Krishnamurthi (2006). Semantics and scoping of aspects in higher-order languages. *Sci. Comput. Program. 63*(3), 207–239.

Gal, A., W. Schröder-Preikschat, et O. Spinczyk (2001). Aspectc++ : Language proposal and prototype implementation. In *Workshop on Advanced Separation of Concerns in Object-Oriented Systems — OOPSLA 2001*.

Havinga, W., I. Nagy, et L. Bergmans (2006). An analysis of aspect composition problems. In *In Proceedings of the 3rd European Workshop on Aspects in Software (EIWAS) 2006*.

Hirschfeld, R. (2003). AspectS — aspect-oriented programming with squeak. In *Proceedings NODe 2002*, Volume 2591 of *LNCS*, pp. 216–232. Springer-Verlag.

Hirschfeld, R., P. Costanza, et O. Nierstrasz (2008). Context-oriented programming. *Journal of Object Technology 7*(3).

Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, et W. G. Griswold (2001). An overview of AspectJ. In *Proceeding ECOOP 2001*, Number 2072 in LNCS, pp. 327–353. Springer Verlag.

Klaeren, H., E. Pulvermüller, A. Raschid, et A. Speck (2000). Aspect composition applying the design by contract principle. In *Proceedings of the 2nd International Symposium on Generative and Component-Based Software Engineering (GCSE 2000)*, Volume 2177 of *LNCS*, pp. 57–69. Springer-Verlag.

Lopez-Herrejon, R., D. Batory, et C. Lengauer (2006). A disciplined approach to aspect composition. In *PEPM '06 : Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, New York, NY, USA, pp. 68–77. ACM Press.

Marot, A. et R. Wuyts (2008). Composability of aspects. In *SPLAT '08 : Proceedings of the 2008 AOSD workshop on Software engineering properties of languages and aspect technologies*, New York, NY, USA, pp. 1–6. ACM.

Nagy, I., L. Bergmans, et M. Aksit (2005). Composing aspects at shared join points. In A. P. Robert Hirschfeld, Ryszard Kowalczyk et M. Weske (Eds.), *Proceedings of International Conference NetObjectDays, NODe2005*, Volume P-69 of *Lecture Notes in Informatics*, Erfurt, Germany. Gesellschaft für Informatik (GI).

Privat, J. et R. Ducournau (2005). Raffinement de classes dans les languages à objects statiquement typés. In *Proceedings of LMO'05*, pp. 17–32. Hermes.

Rajan, H. et K. Sullivan (2003). Eos : instance-level aspects for integrated system design. *SIGSOFT Softw. Eng. Notes 28*(5), 297–306.

Tanter, É. (2006). Aspects of composition in the Reflex AOP kernel. In W. Löwe et M. Südholt (Eds.), *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, Volume 4089 of *LNCS*, Vienna, Austria, pp. 98–113. Springer.

Tanter, É. (2008). Expressive scoping of dynamically-deployed aspects. In *Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development (AOSD 2008)*, Brussels, Belgium, pp. 168–179. ACM Press.

Summary

Aspects interaction and composition is a hot topic in the aspect-oriented community. Although some solutions are available (*e.g.*, advice and aspect ordering), approaches proposed by widely used aspect languages assume that a global knowledge of the aspects is a prerequisite in order to compose them. This does not help in avoiding implicit inter-aspects interactions.
In this paper we present aspectboxes, a visibility mechanism for aspects. An aspectbox, an

elementary unit of visibility, encapsulates definitions of aspect. An aspectbox may be used by other aspectboxes in order to favor incremental software refinement. A class may use an aspectbox to benefit from the aspects defined in it.